INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Multiple-Block Ahead Branch Predictors

André Seznec, Stéphan Jourdan, Pascal Sainrat, Pierre Michaud

N° 2825

Mars 1996

THÈME 1

ISSN 0249-6399

Multiple-Block Ahead Branch Predictors

André Seznec, Stéphan Jourdan, Pascal Sainrat, Pierre Michaud*

Thème 1 — Réseaux et systèmes

Projet Caps

Rapport de recherche n° 2825 — Mars 1996 — 27 pages

Abstract: A basic rule in computer architecture is that a processor cannot execute an application faster than it fetches its instructions. To overcome the instruction fetch bottleneck shown in wide-dispatch "brainiac" processors, this paper presents a novel cost-effective mechanism called the multiple-block ahead branch predictor that predicts in an efficient way addresses of multiple basic blocks in a single cycle. Moreover and unlike the previous multiple predictor schemes, the multiple-block ahead branch predictor can use any of the branch prediction schemes to perform very accurate predictions required to achieve high-performance on superscalar processors. Finally, we show that pipelining the branch prediction process can be done by means of our predictor for "speed demon" processors to achieve higher clock rate.

(Résumé : tsvp)

This work was partially supported by GDR AMN (MIF project).

Stéphan Jourdan and Pascal Sainrat are with the APARA team at IRIT, Toulouse, France.

This report is also published as IRIT report No 96-09-R

*{seznec,pmichaud}@irisa.fr,{sainrat,jourdan}@irit.fr}

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Une prédiction de branchement plusieurs blocs en avance

Résumé : Un processeur ne peut exécuter plus vite qu'il ne charge ses instructions. Pour permettre de réduire le goulôt d'étranglement noté sur les processeurs à nombreuses unités fonctionnelles, cet article présente un mécanisme appelé "multiple-block ahead branch predictor" qui prédit efficacement les adresses de plusieurs blocs d'instructions en avance. Un avantage majeur de notre mécanisme, est qu'il peut être utilisé avec à peu près tous les schémas de prédiction de branchement. Nous montrons de plus que l'utilisation de ce schéma permet de pipeliner la prédiction de branchement

Multiple-Block Ahead Branch Predictors

Abstract

A basic rule in computer architecture is that a processor cannot execute an application faster than it fetches its instructions. To overcome the instruction fetch bottleneck shown in wide-dispatch "brainiac" processors, this paper presents a novel cost-effective mechanism called the multiple-block ahead branch predictor that predicts in an efficient way addresses of multiple basic blocks in a single cycle. Moreover and unlike the previous multiple predictor schemes, the multiple-block ahead branch predictor can use any of the branch prediction schemes to perform very accurate predictions required to achieve high-performance on superscalar processors. Finally, we show that pipelining the branch prediction process can be done by means of our predictor for "speed demon" processors to achieve higher clock rate.

1 Introduction

With the advance of technology, current superscalar processors can fetch and execute several instructions in parallel on each cycle. For instance, the MIPS R10000 [18] fetches up to four instructions and executes up to five instructions per cycle by means of five functional units. Recent studies [14] have shown that the potentially available *instruction-level parallelism* (ILP) in general-purpose integer applications is higher than six instructions per cycle while assuming a perfect instruction-fetch mechanism.

To best exploit the available ILP, wide-dispatch processors featuring a dozen functional units working in parallel would be required. The HP PA-8000 [7] has already been implemented with a comparable number of functional units. Unfortunately, the instruction-fetch mechanisms implemented in current commercial microprocessors do not fully exploit this potential parallelism. For these processors, the instructions fetched in a single cycle must belong to the same basic block, and are not usually permitted to span two cache lines. Since a processor cannot execute instructions faster than it fetches them, these constraints significantly impair performance, particularly on codes featuring many small basic blocks. One way to partially remove the instruction fetch bottleneck, is to fetch instructions belonging to

multiple consecutive basic blocks, as is done in several processors such as the POWER2 [26]. In order to solve the whole problem, multiple non-consecutive basic blocks must be fetched in a single cycle as most basic blocks are only four to six instructions long. A processor featuring such a mechanism would have to predict multiple targets and branch outcomes in a single cycle. Several recent papers have proposed implementations for such a predictor [27] [4]. These studies have established the viability of predicting two branches in a single cycle, but are not attractive because of the large hardware costs required and the prediction scheme used. One should note that another way to solve this problem is to enlarge the size of the basic blocks by means of such techniques as predicated execution [16] or super-block scheduling [10]. Despite the fact that this paper focuses on hardware schemes, the two approaches are not exclusive.

In superscalar processors, blocks of consecutive instructions are fetched in parallel. The last instruction of such a block is either a branch or is determined by some implementation constraints (for instance, the boundary of a cache block or the maximum number of instructions in the block). Throughout the paper, we refer to processors that can fetch only one basic block per cycle as *single I-fetch processors*, to processors that can fetch two non-consecutive blocks per cycle as *double I-fetch processors*, and to *Multiple I-fetch processors* as an extension to the latter case. In this paper, we first compare the performance of single I-fetch processors and multiple I-fetch processors with respect to various parameters such as the prediction accuracy rate and the instruction-dispatch buffer depth. Assuming equivalent branch prediction accuracy, our simulations point out that double I-fetch processors significantly outperform single I-fetch processors when the dispatch width is wider than four instructions per cycle, motivating the need for fetch mechanisms that can fetch two non-consecutive blocks in parallel. The simulations also show that given current instruction-set architectures and compiler technology, fetching more than two non-consecutive blocks per cycle does not significantly improve performance.

We then propose a complete and cost-effective mechanism called the *Two-Block Ahead Branch Predictor*, that predicts the addresses of two blocks per cycle. This mechanism consists of a two-block ahead branch prediction table, a two-block ahead branch target buffer and a two-block ahead return stack. Such an approach can obviously be extended to

fetch more than two non-consecutive instruction blocks in a single cycle; we refer to this as a multiple-block ahead branch predictor.

In conventional branch prediction mechanisms, information associated with the current instruction block such as its memory address, is used to predict the next instruction block. Previous multiple predictors rely on the same information to predict the two subsequent instruction blocks. The originality of our mechanism is to use information associated with the current instruction block to predict the block following the next instruction block. Thus the two-block ahead predictor uses both fetch addresses to predict the two subsequent instruction blocks to fetch on the next cycle. The amount of information stored in the two-block ahead branch predictor is in the same range as in a conventional branch prediction mechanism. Moreover, any prediction scheme can be used to determine the outcome of conditional branches leading to better prediction accuracy.

As described in [5], two different approaches are used to achieve high performance in superscalar machines: "brainiacs vs. speed demons". The two-block ahead predictor is really useful in double I-fetch "brainiac" processors since it increases the amount of potential ILP. Furthermore, it can also be used effectively in single I-fetch "speed demon" processors to increase the clock rate. Indeed in current microprocessors, either the branch prediction is completed in a single cycle, or pipeline bubbles are inserted on each predicted taken branch. The predictor is a critical path in the processor. Implementing the two-block ahead branch predictor in a single I-fetch "speed demon" processor would remove this critical path by allowing the branch predictor to be pipelined.

The remainder of the paper is organized as follows. Related work is discussed in section 2. Section 3 details the model of execution and introduces the simulation process used throughout the paper. Section 4 compares single I-fetch and multiple I-fetch processors. Section 5 introduces the two-block ahead branch predictor. Section 6 shows that the branch prediction process can be pipelined in single I-fetch "speed demon" processors by means of our two-block ahead branch predictor. Finally, simulation results are reported in section 7. Section 8 concludes the paper.

2 Related Work

To our knowledge the pipelining of the branch prediction has never previously been addressed. Only a few studies [27, 19, 3] have addressed the problem of fetching multiple non-consecutive basic blocks in a single cycle.

In order to fetch two basic blocks in a single cycle, Yeh et al. [27] proposed to store 6 addresses in each entry of their *Branch Address Cache* (BAC): T, N, TT, TN, NT, and NN, where N and T refer to the outcome of the branches (*not-taken* and *taken* respectively). The branch prediction mechanism can predict two branches in a single cycle. According to the prediction made by a global scheme (address-based schemes give lower prediction accuracy since they use the same address for both predictions), the addresses of the two subsequent basic blocks are returned with a hit in the BAC. When a branch is resolved for the first time, an entry is allocated in the BAC with fields T and N set (primary fields). If the previous fetch address had a valid primary branch entry in the BAC with secondary fields cleared, and if there was enough bandwidth to fetch another basic block, then T and N are also inserted in these secondary fields. This introduces wasted fields when entries are allocated for primary branches with not enough fetch bandwidth left for a second basic block (with a dual-ported instruction cache, this occurs each time a basic block lies across an aligned block boundary). Finally, a basic block can belong to several BAC entries, so their mechanism does not require any static partitioning. Redundancies are however created, but the scheme does not rely on any compiler work.

In [19], the authors proposed to split the *Control Flow Graph* (CFG) into subgraphs. In order to fetch two non-consecutive basic blocks even when they belong to different cache lines, they use tree-like subgraphs of depth 3 [4]. Nodes of the subgraphs are straightline pieces of code (basic blocks). Intermediate nodes (depth 0 and 1) can be terminated by any control-changing instructions while the last nodes (depth 2) are terminated by single-target instructions (either unconditional branches, procedure calls, or non-branch instructions), and their lengths are limited by the instruction-fetch bandwidth. Intermediate outcomes are not predicted. Instead, one path is predicted in the subgraph among four. All parameters required to describe a subgraph are stored in a Subgraph History Table (SHT). In order

to have no redundant information in the SHT, each basic block should belong to only one subgraph. However since each entry in the SHT holds a rigid subgraph structure, there might be many wasted fields. Indeed, a static CFG is not as simple as a tree and it cannot be perfectly partitioned into tree-like structures of depth 3. The prediction mechanism gives good accuracy results compared to non-hybrid schemes without a lot of additional logic. But their scheme mostly relies on compiler work to partition the CFG into tree-like subgraphs. This approach may be a good alternative to conventional branch prediction schemes in the future.

In [3], the authors introduced a mechanism called the *Collapsing Buffer* that achieved *merging* [13]. This mechanism can fetch multiple basic blocks in a single cycle as long as they belong to the same cache line, and otherwise performs some alignments between two basic blocks by means of a pipelined fetch mechanism (*banked sequential*). The scheme features an interleaved coupled BTB/BHT providing one entry to each instruction of a cache line. The Collapsing Buffer scheme is efficient as long as branch targets address the same cache line, and performs well on their execution model retiring less than 2.5 instructions per cycle on integer applications. Another major drawback is the requisite use of a 2-bit counter prediction scheme [22] which results in a higher misprediction rates than other schemes such as local schemes [17][28]. Moreover, as the I-cache line size keeps growing in current processors, the interleaving factor of the BTB grows as much and the collapsing logic becomes more complex. Although this approach is interesting, our purpose is to go one step beyond by fetching multiple basic blocks in a single cycle, even when they belong to different cache lines. One should note that our approach is not incompatible with collapsing.

3 Experimental Setup

3.1 Traces

The experimental results presented in this paper are based on the programs from the SPEC92 suite [25]. Two subsets are defined: CINT92 (*integer*) and CFP92 (*floating-point*). All programs from both sets were used to make our evaluations. The benchmarks were compiled on a R4600-based SGI workstation using `cc` and the standard makefiles provided with the

suite (with all optimizations turned on). We used the PIXIE profiler [24] to collect instruction traces from a real processing of the SPEC benchmarks, including library calls. These traces fed our simulator which performs a *cycle-by-cycle* simulation and gathers the mean number of *instructions retired per cycle* (IPC).

All the benchmarks were run to completion except for *backprop*, *dnasa7*, *wave5*, and *spice2g6*. Due to time constraints, the smallest input files or slightly modified versions have been used. In all, more than 600 million instructions have been captured (with all NOPs removed from the traces).

3.2 Machine Model

The modeled architecture depicted in figure 1, implements out-of-order and speculative execution policies in order to best exploit ILP. In brief, after being fetched, instructions are decoded and dispatched in-order from the instruction-dispatch buffer to the instruction-issue buffer. The upper bound of the number of instructions dispatched each cycle defines the dispatch width. Registers are renamed using a map table during the dispatch process. Instructions in the issue buffer may be issued out-of-order when all their operands are available, and a *max-dependent* selection mechanism as described in [1] is used when more than one instruction compete for the same functional unit access. To enforce precise interrupt management, a history buffer [23] similar to the active list of the R10000, records the previous mappings discarded by the renaming process during the dispatch stage. Checkpoints [9] of the map table (architectural state) are established at every branch in order to recover from branch misprediction in one cycle, regardless of the number of mapping modifications recorded in the history buffer. With such a scheme, the history buffer is only used to recover from other exceptions and to keep track of the state of the physical registers in order to free them when the instructions complete. When an instruction finishes execution, its result updates the processor state but its corresponding entry remains in the history buffer until all previously dispatched instructions can no longer produce interrupts. Instructions capable of generating interrupts are conditional and indirect branches, divides, and memory accesses. In the latter instruction class, subsequent entries may be dequeued as soon as the address

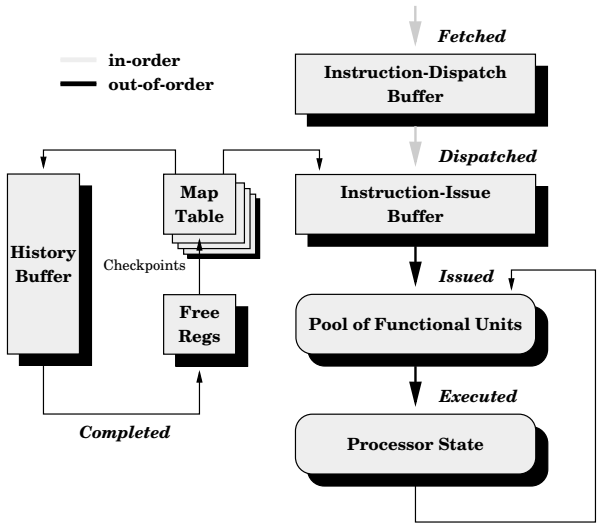


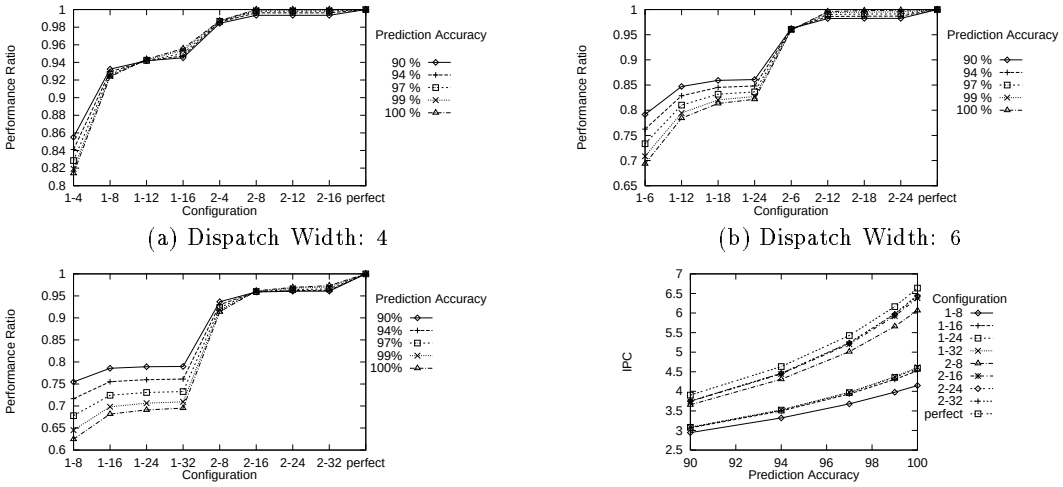
Figure 1: Simulated Out-of-Order Microarchitecture.

is computed. Each cycle, multiple out-of-order instruction retirements can be made, freeing the physical registers to be reused in the renaming process.

A previous study [15] has shown that configurations reported in table 1 of such out-of-order architectures give almost no performance loss over perfect configurations only limited by the size of the lookahead window, assuming an ideal-fetch mechanism and no misprediction. The instruction latencies used in the simulations were those of the PowerPC 604 [11]. The mean-IPC values varied from 3.6 to 6.5 on integer programs according to the dispatch width (4, 6, and 8 instructions dispatched per cycle). We keep their configurations (DW 4, DW 6 and DW 8) in order to evaluate the fetch mechanisms. The lookahead window is the maximum number of dispatched instructions that can be processed at the same time, sometimes referred as the instruction window. Moreover, we assumed for all the models a unified issue-buffer, and a maximum number of 16 checkpoints as in the Sparc64 [20]. Such a value does not degrade the performance of any of the models. Finally, one should note that our results do not rely on any misfetch or mispredict penalty value since the simulator models a real processor except for the data cache which is modeled as an infinite cache.

Parameters	DW 4	DW 6	DW 8
Dispatch Width	4	6	8
Lookahead Window Size	32	64	96
Issue Buffer Depth	28	48	72
Number of Fixed-Point Units	3	4	5
Number of Floating-Point Units	2	2	2
Number of Branch Units	2	2	3
Number of Data-Cache Ports	2	3	4

Table 1: Model Configurations.



(a) Dispatch Width: 4

(b) Dispatch Width: 6

(c) Dispatch Width: 8

(d) Dispatch Width: 8

Figure 2: Performance of Single and Multiple I-Fetch Processors.

4 Single vs. Multiple I-Fetch Processors

While multiple basic block fetch mechanisms have already been proposed, there was no clear study showing that such mechanisms would provide performance enhancements. The purpose of this section is to show that despite data dependencies and resources hazards, multiple fetch mechanisms would give significant performance improvement in wide-issue out-of-order processors.

In most processors, the fetch mechanism consists mainly of three parts: an instruction cache from where the instructions are fetched, an instruction-dispatch buffer where the instructions are maintained waiting to be dispatched, and some branch prediction structures predicting the outcome and the target address of any fetched branch. The dispatch buffer decouples the instruction fetching from the dispatch process, sustaining a better throughput in the presence of cycles in which only a small number of instructions can be fetched (the buffer can be filled in a single cycle).

Multiple I-fetch is useless with long basic blocks. The CFP92 subset features long basic blocks close to fifteen instructions long on average. Simulations on floating-point programs, not reported here, have shown that a single I-fetch processor is effective whatever the dispatch width may be provided a deep dispatch buffer. A 8-wide single I-fetch processor gives over 97.3% of the perfect performance when the prediction accuracy is higher than 90%.

Double I-fetch is useful with small basic blocks. The average size of the basic blocks in the CINT92 subset is five instructions long. Figures 2 (a) through (d) show the results on integer programs according to the depth of the instruction-dispatch buffer, the prediction accuracy, and the dispatch width. Only the geometric means of the results are reported. Note that we assume in these simulations a perfect I-cache and no alignment problem due to cache line boundaries. Configurations are based both on the number of basic blocks fetched in a single cycle, and on the depth of the dispatch buffer. For instance, configuration 2-8 denotes a double I-fetch processor featuring a 8-deep dispatch buffer.

In (a), (b), and (c), the ratio gives the relative performance between the evaluated configuration and a perfect fetch mechanism with the prediction accuracy remaining the same. The curves clearly state that 4-wide processors do not require any improvement over a single I-fetch policy except for a 8-deep dispatch buffer, giving 93% of the perfect performance whatever the prediction accuracy may be. On the other hand, 6-wide and especially 8-wide double I-fetch processors give a huge improvement of performance over single I-fetch processors. In a 8-wide processor, the improvement is between 20 and 40% depending on the prediction accuracy, assuming a deep dispatch buffer. Moreover these

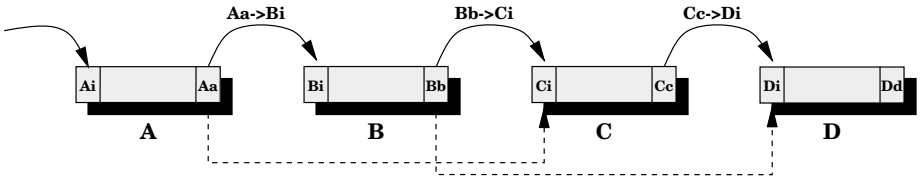


Figure 3: Two-Block Ahead Branch Information

results bring to light that fetching more than two basic blocks in a single cycle is not effective as double I-fetch mechanisms provide nearly 100% of performance. One should note that the relative benefit when fetching two basic blocks in a single cycle increases with the branch prediction accuracy. Finally, an instruction buffer twice as big as the dispatch width is required in any case.

As shown in figure (d) for a 8-wide processor, it is more effective to increase the number of blocks fetched per cycle than to improve the prediction accuracy. Nevertheless, these two optimizations are not exclusive and they each give new opportunities for performance improvement.

5 The Two-Block Ahead Branch Predictor

As stated in the introduction, the two-block ahead branch predictor uses information associated with the current instruction block to predict the address of the instruction block that is two blocks ahead. Its principle is illustrated in Figure 3 where Ai, Bi, Ci, and Di are the basic block starting addresses and Aa, Bb, Cc, Dd are the branch addresses. Any of the control-flow transitions can be fall-through. While the instruction blocks A and B are fetched, the two-block ahead branch predictor uses Aa instead of Bb to predict Ci, and Bb to predict Di. The behavior of each part of the predictor is described in the following sections. Figure 4 presents an implementation of the Two-Block Ahead Branch Predictor for a double I-fetch processor.

5.1 The Two-Block Ahead Branch Prediction Table

Instead of using the address of the conditional branch to predict its outcome, our scheme always uses the address and the history register of the previous branch ((Aa,Ha) and (Bb,Hb) to predict Ci and Di respectively). Such a scheme can be adapted to use any branch prediction schemes combining address and history (see for instance [17, 27]).

We will show in the next section that the prediction is as accurate as if the address of the branch had been used instead. In figure 4, Pa and Pb refer to the predicted outcome when the Branch Prediction Table (PT) is indexed with Aa and Bb respectively.

5.2 The Two-Block Ahead Branch Target Buffer

BTB entry description. The two-block ahead BTB records information for a given branch in an entry associated with both the address of the previously fetched block and the type of transition between both blocks. When a taken branch Bb is mispredicted or misfetched, a BTB entry is allocated to record its target (Ci), type (conditional, unconditional, indirect, or return) i.e. 2 bits, and position *b* in line B (for instance, 4 bits for a 16 instruction cache line size). The BTB is indexed for that entry with: (1) the address of A, (2) the position *a* of the branch in A, and (3) the type of the transition between Aa and Bi (Aa→Bi) (2 bits). If no branch was fetched with A, *a* would be the last instruction in line A. There are three types of transition: T (Aa is a non-return taken branch), N (Aa is a non-taken conditional branch or a non-branch instruction), and R (Aa is a return). Type R is special and is further explained when introducing the procedure return mechanism.

BTB read. In order to predict two addresses in a single cycle in double I-fetch processors, both addresses Ai and Bi are used to access the BTB in addition to the I-cache. Both the branch position *a* and the transition X between Aa and Bi were determined during the previous cycle. We can therefore check for BTB entry AaX to compute Ci. If AaX misses then we assume no branch in B, and *b* would be the last instruction in line B. Otherwise, the BTB entry AaX holds the target Ci of branch Bb, its type, and its position *b*. If Bb is a conditional branch, the outcome is provided by the two-block ahead branch prediction table. All the information required to compute Ci is known and the process is detailed in

figure 4. The address generator also produces b and the transition Y between Bb and Ci. Once these values are produced, the tag checking for the BTB entry BbY begins and the address Di is computed in a similar way. So indexing the BTB with A and B is done in parallel, but the tag-matching for AaX and BbY is partially serialized. This constraint is part of the parallelism vs. speed tradeoff ("brainiacs vs. speed demons").

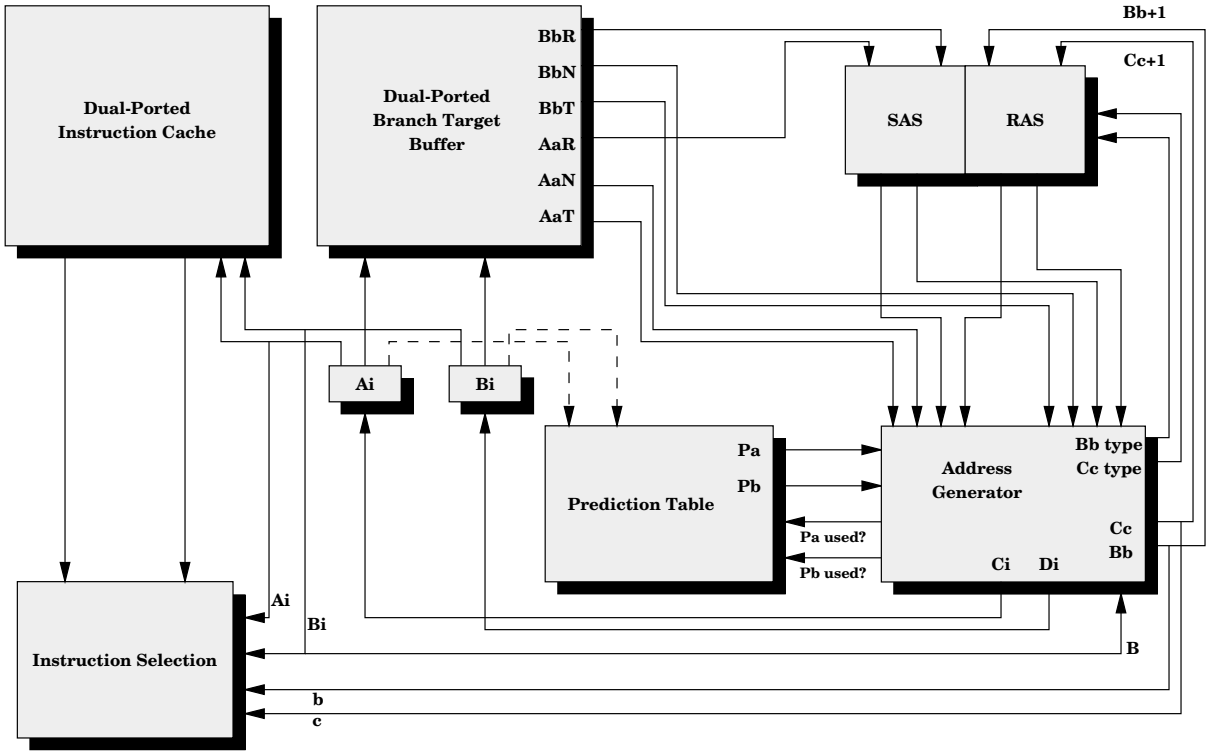
Hardware cost. The amount of information stored in a two-block ahead BTB entry is only a few bits wider than in a conventional BTB (position b of the branch in B and the transition type Aa→Bi). Two entries may be associated with a single address Aa. The BTB entry AaT records information for a branch in the target basic block of branch Aa, and entry AaN records information for a branch in the fall-through basic block of branch Aa. Entry AaX identifies a single basic block, hence a single target, except for AaT when Aa is an indirect branch or a procedure return. Hence, the two-block ahead BTB does not require more entries to achieve the same hit ratio as a conventional BTB.

Associativity. Since most of the conditional branches are either mostly taken or mostly fall-through [28], the BTB will often only record one of AaT or AaN. Thus the associativity required in the two-block ahead BTB is not higher than in a conventionnal BTB. An entry AaX is mapped in the BTB as follows: low-order bits of A are used to address the set (BTB indexing), and both aX and high-order bits of A are used to tag the entry allocated within the set.

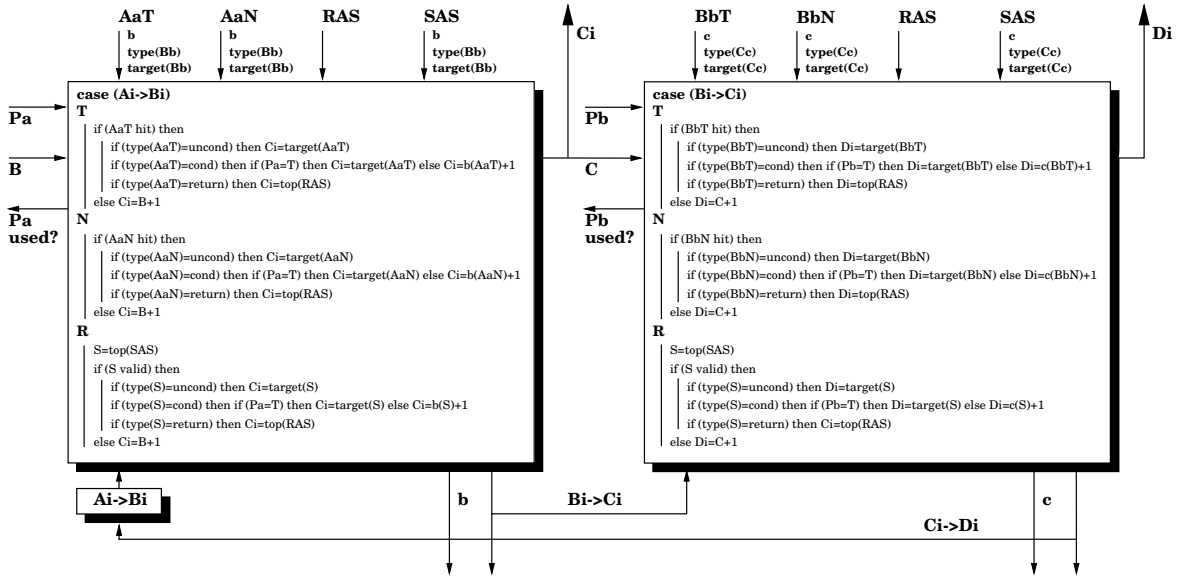
5.3 Coping with Procedure Returns and Indirect Jumps

As done in most processors to get the address for procedure returns, the two-level branch predictor relies on a Return Address Stack onto which return addresses are pushed when the calls are fetched. Nevertheless, an entry AaX identifies a single basic block except when Aa is an indirect branch or a return. In the case of returns, several branches could conflict for the same entry indexed by the return address as mentioned in [27].

To overcome this problem, Pp being the procedure call and Bb the procedure return in the example shown in figure 5, a BTB entry PpR is allocated instead of an entry BbT



(a) Predicting Two Basic Blocks



(b) Instruction Address Generation

Figure 4: Two-Block Ahead Branch Predictor in Double I-Fetch Processor.

on a misprediction when there is a branch in the block C to keep information about the branch Cc (fig. 5 (c)). The BTB is searched for an entry PpR whenever a call instruction Pp is fetched (fig. 5 (a)). In order to maintain this information until the return instruction fetching, the two-block ahead branch predictor uses a Second Address Stack (SAS). A copy of the PpR entry is pushed into the SAS whenever PpR hits in the BTB. Otherwise, an invalid entry is pushed. When the return is predicted to be fetched the next cycle, the entry is popped out from the SAS in order accurately predict a branch in the subsequent basic block if any (fig. 5 (b)).

The whole process is further detailed in figure 4.

Finally indirect branches are predicted in the two-block ahead branch predictor by recording their last target addresses. Therefore an entry AaT might be allocated when Aa is an indirect jump in order to predict the branch in block B if any.

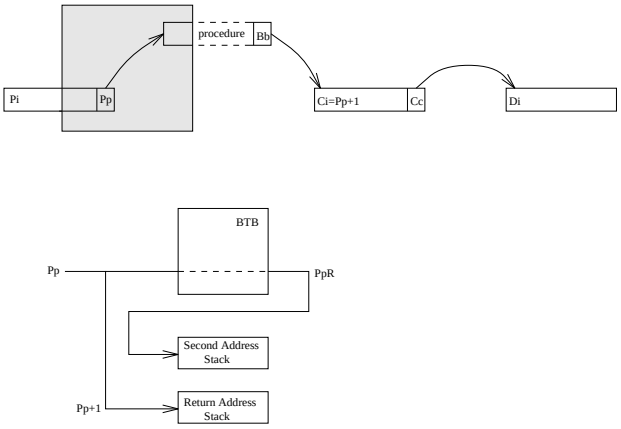
6 Single I-Fetch Processors and Two-Block Ahead Branch Prediction

6.1 Instruction Address Generation may be a Critical Path

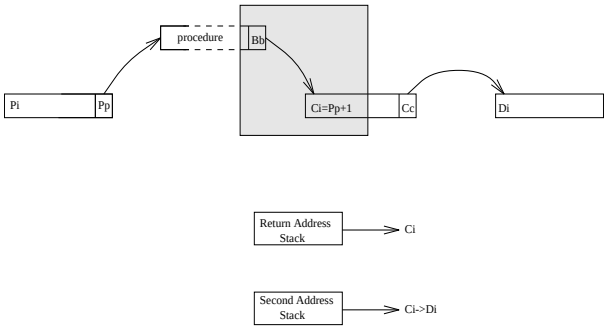
In current single I-fetch processors, both the I-cache and the branch predictor are accessed with the current instruction block starting address. By the end of the cycle, the starting address of the next instruction block must be generated. In some of the processors, the I-cache access time is longer than the cycle time, leading to a pipeline structure depicted in figure 6 (a)[1]. For instance, the Intel PentiumPro [8] features a pipelined I-cache access completed within two and a half cycles.

As far as the current instruction block address is used to predict the next instruction block, either the instruction address generator can compute the starting address of the next instruction block in a single cycle as in the PentiumPro, or bubbles are inserted in the pipeline in the case of branches as in the DEC 21164 [6]. The instruction-address generation process is quite complex because it includes several consecutive steps:

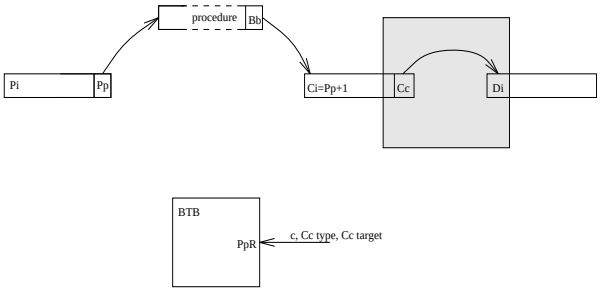
[1] The Instruction Address Generation stage is usually not illustrated in the presentation of commercial microprocessors. The PentiumPro is a notable exception.



(a) - Call Pp is detected at fetch time, the stacks are written



(b) - Ret Bb is detected at fetch time, the stacks are read



(c) - Branch Cc was mispredicted, the BTB is updated

Figure 5: The two-block ahead return stack.

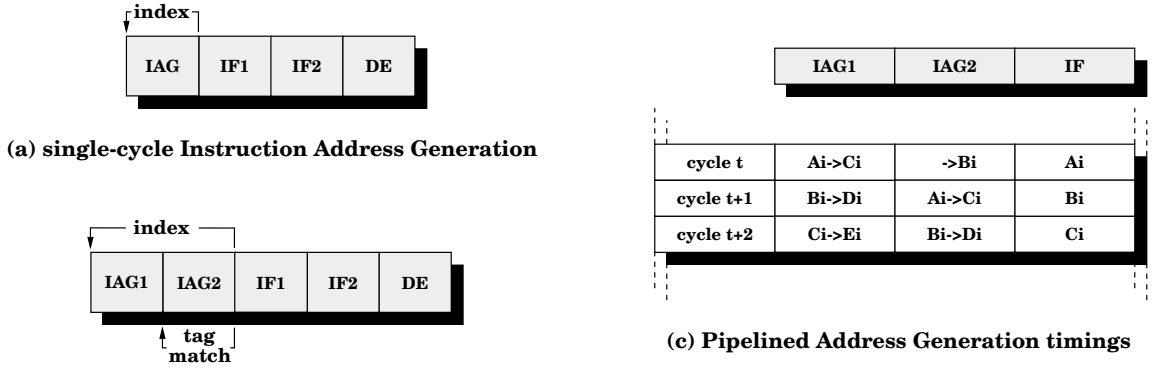


Figure 6: Instruction-Fetch Pipelines.

1. Parallel accesses to the BTB, the Prediction Table (PT), and the return address stack. Computation of the fall-through address.

2. Prediction of the outcome and selection of the generated address. Possible updates of the return stack and the branch history register.

In particular, the read of a set-associative BTB is time consuming. Achieving the complete instruction address generation process in a single cycle may be a challenging problem in high clock-speed processors. The instruction address generator might then be the critical electrical path, determining the processor cycle time.

6.2 Pipelining the Instruction Address Generation Process

In Section 7, we will show that branch prediction information can be associated with the previous branch instructions without degrading the prediction accuracy. With such a scheme, two addresses are predicted in a single cycle in double I-fetch "brainiac" processors as shown in the previous section. Instead of exploiting more parallelism, another way to get performance improvement is to increase the clock rate, leading to single I-fetch "speed-demon" processors. In that case, the instruction-address generator stage may be the critical path of the processor. We show here that pipelining the instruction address generation process (fig.

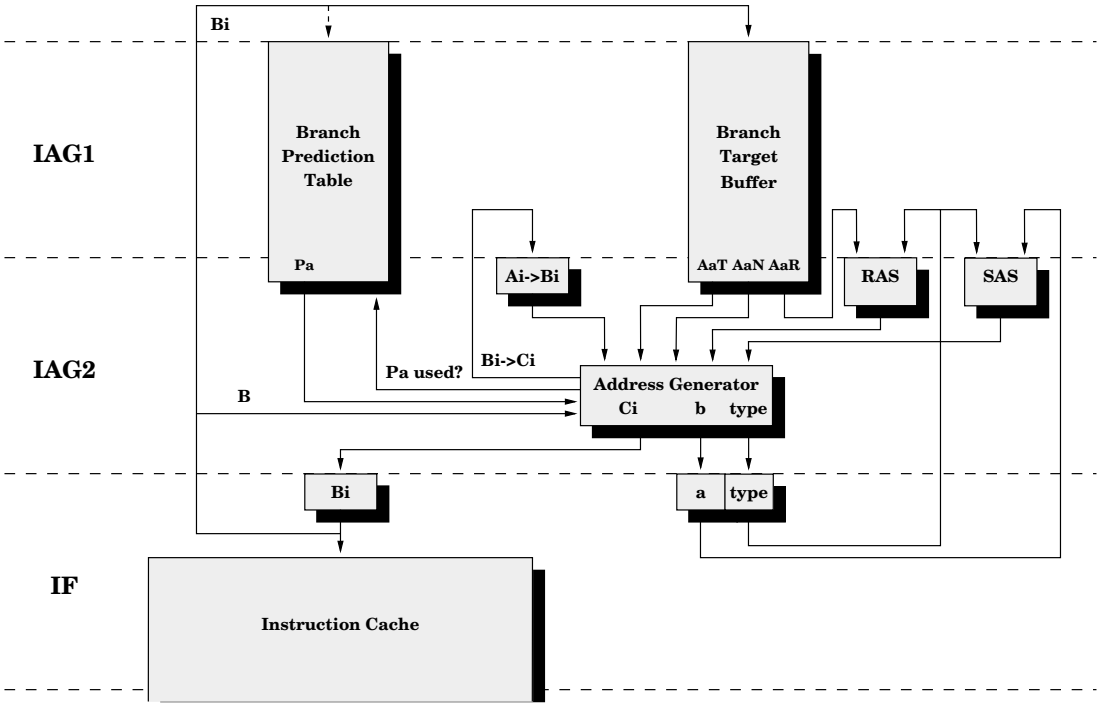


Figure 7: The Two-Stage Pipelined Branch Predictor (cycle t+1).

6 (b)) in such single I-fetch processor can be done by means of the two-block ahead branch predictor.

A two-stage pipelined branch predictor is depicted in Figure 7. The BTB and PT illustrated in this figure are a two-block ahead BTB and PT implementations computing only one address in a single cycle.

Let Ai, Bi, and Ci be the starting addresses of the instruction blocks respectively fetched at cycle t, t+1, and t+2. Ha, Hb, and Hc are respectively the branch history registers during cycle t, t+1, and t+2. The behavior of the pipelined address generator is represented in figure 6 and is as follows:

1. **cycle t**: In stage IAG1 of the pipeline, Ai and Ha are used to index the PT and the BTB. In stage IF, the fetching of the instruction block A begins. At the end of the

cycle, Bi and a flow out from the stage IAG2 as well as the type of the transition from Ai to Bi (T, N, or R).

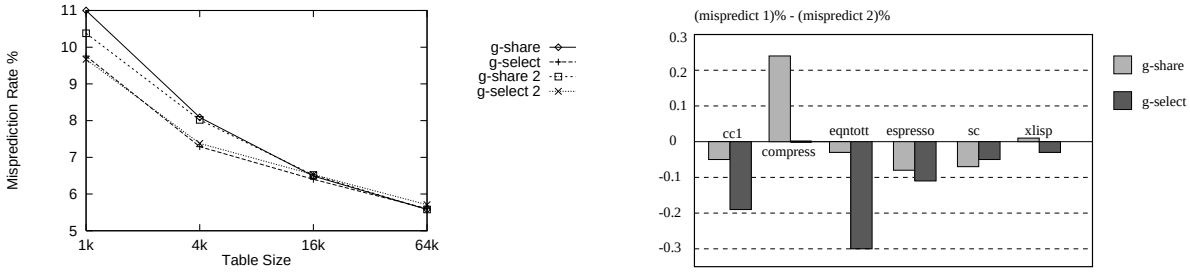
2. **cycle t+1**: In stage IAG1, Bi and Hb are used to index the PT and the BTB while the fetching of B begins in stage IF. In stage IAG2, the access to the BTB and the PT with Aa are completed. The fall-through address for block B is also computed (Bb+1). Depending on the type of the transition from Ai to Bi, and on the information flowing out from the BTB and the PT, Ci is chosen among 5 addresses (the target addresses contained in AaT, AaN, top of the RAS or top of the SAS, or the fall-through address Bb+1) as related in the first algorithm of figure 4. Before the end of cycle t+1, the position b is forwarded to stage IAG1 to proceed the tag-matching process.

3. **cycle t+2**: Ci is used to index the cache in stage IF and to compute the two-block ahead instruction block starting address in stage IAG1.

One should note that some information (address Bi, type of the transition from Ai to Bi) produced during cycle t are used during cycle t+1. Nevertheless these items are not critical. Basically, during the instruction address generation process, the most time consuming actions are the accesses to the BTB and the PT. Therefore pipelining the instruction address generation process as described above allows the use of a shorter cycle than with conventional address generators.

Misprediction penalty. Using a two-block ahead branch predictor in an out-of-order single I-fetch processor does not result in a one-cycle increase of the misprediction penalty. As a matter of fact, the address of the non-predicted path is recorded in the checkpoint established for the branch to resume fetching in processors featuring a one-block ahead branch predictor. The two-block ahead scheme only requires to record in addition the non-predicted path in the IAG2 stage (AaT or AaN if branch Aa was mispredicted) and the prediction made Pa, since all the other information required in stage IAG2 to compute Ci would be known (fall-through address and return stack values).

In in-order single I-fetch processors, a structure can hold such values. Otherwise, the misprediction penalty would be increased by one cycle. This extra cycle is required to retrieve



(a) Impact of the size of the PT

(b) Individual Results with a 64k-entry PT

Figure 8: Branch Misprediction Rates.

both the prediction and the possible addresses of the target instruction Ci. Nevertheless, one should note that replacing an instruction address generator resulting in pipeline bubbles on branches as in the DEC 21164 or the MIPS R10000 by our predictor would save the bubble in all cycles where the prediction is correct. While on incorrect predictions, the penalty is the same for both mechanisms.

7 Experimental Results

Trace-driven simulations were conducted to verify the effectiveness of the two-block ahead branch predictor. We first establish that the branch prediction accuracy achieved by our branch prediction table is equivalent to those obtained with conventional one-block ahead branch prediction tables. Finally, we show that the two-block ahead branch predictor does not require any additional BTB logic to handle most branches.

7.1 Branch Prediction Accuracy

We assumed a perfect BTB (all branches hit) in these simulations to compare predictors without any clouding effects from the BTB. The simulations were run only over the CINT92 suite since floating-point benchmarks tend to lower isprediction rates.

Figure 8 (a) presents the average misprediction rate for two common prediction schemes with respect to the size of the prediction table. The misprediction rates are reported for both the two-block ahead branch predictor (*g-share 2*, *g-select 2*) and the corresponding

one-block ahead branch predictor (*g-share*, *g-select*). These branch prediction schemes differ by the index which is used to access the prediction table. The prediction table in *g-select* is indexed with a concatenation of branch history and branch address bits. The index value in *g-share* is the exclusive OR of the branch address with the branch history register.

It is noticeable that, for both schemes and for all table sizes, the performance of the two-block ahead branch predictors is very close to the performance of the corresponding one-block ahead branch predictors. The difference between the misprediction rates of the different benchmarks are reported in figure 8 (b) for a 64 K-entry prediction table. These differences are very tiny and do not exceed 0.30 %.

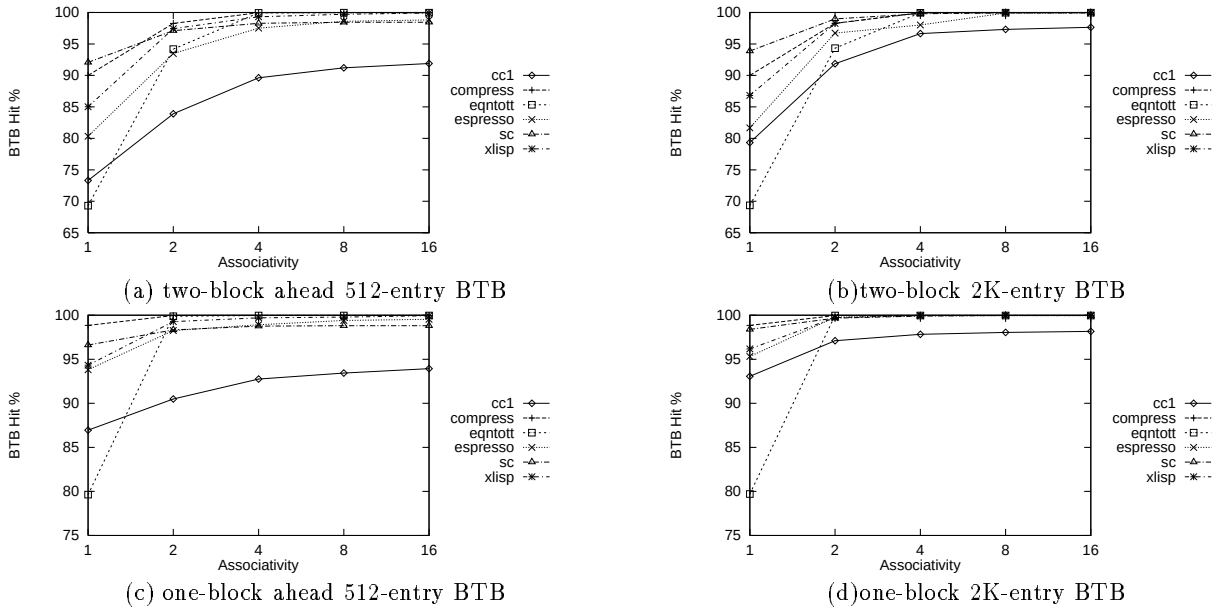
From these simulation results, we conclude that the two-block ahead branch history register and the two-block ahead address are as representative of a branch as the conventional branch history register and the branch address.

7.2 The Branch Target Buffer

In the previous sections, we have introduced the two-block ahead branch target buffer to predict two block addresses per cycle or to pipeline the address generation process. Here our results verify that such a mechanism does not require a high degree of associativity or a large number of entries since a branch can be associated with more than one BTB entry.

Figure 9 reports the individual results with a 512-entry and a 2K-entry BTB according to associativity. A pseudo-random replacement policy was used and the cache line size was assumed to be 16 instructions wide as in the MIPS R10000, the HaL Sparc64, and the PowerPC 620. All the branches in the same cache line map to the same set in the BTB. In addition in the two-block ahead BTB, different types of branches may be associated to the same address tag (AaT and AaN for instance). Thus a set-associative BTB is required.

We can see from figures 9 (a) and 9 (b) that nearly the maximum hit rate is reached with an associativity of 4, compared to an associativity of 2 for a conventional BTB (figures 9 (c) and 9 (d)). These results also show that for realistic BTB sizes, the hit rate for a conventional BTB is slightly better than that for a two-block ahead BTB. But this difference is less than 0.5 % for most applications. So the improvement of fetching two blocks per cycle is still valuable.



(a) two-block ahead 512-entry BTB

(b)two-block 2K-entry BTB

(c) one-block ahead 512-entry BTB

(d)one-block 2K-entry BTB

Figure 9: BTB Hit Rates According to the Associativity and the Number of Entries.

8 Summary and Concluding Remarks

The current instruction-fetch mechanisms limit the performance that may be achieved. New solutions have to be implemented in next generation microprocessors

Two design philosophies have been used to achieve higher performance for the past four years. "Brainiac" processors attempt to achieve the highest level of IPC possible. We have shown that future generation "brainiac" processors should fetch more than one basic block in a single cycle, otherwise the fetch limit of one basic block per cycle would significantly impair the performance. This raises the difficult issue of predicting multiple instruction blocks in parallel. On the other hand, "speed demon" processors get high-performance by stressing the clock rate. All parts of the processor must be pipelined and some functions are spread over several cycles (e.g. I-cache access). But the address generation process is not pipelined in current designs. In these processors, either the address generation mechanism (including branch prediction) becomes the electrical critical path or pipeline bubbles are inserted on

each predicted taken branch. As this may severly limit the performance achieved in future designs, pipelining the address generation and the branch prediction is also a major issue.

We have introduced The Two-Block Ahead Branch Predictor in this paper to deal with both issues. In conventional branch prediction mechanisms, information associated with the current instruction block such as the address of the branch instruction is used to predict the next instruction block. The two-block ahead branch predictor uses the same information when predicting the block following the next instruction block. The amount of information stored in our predictor is in the same range as in a conventional branch prediction mechanism. Furthermore, any branch prediction schemes proposed for single I-fetch processors can be adapted to our predictor. Simulations have shown that equivalent branch prediction accuracy is achieved. Thus, the two-block ahead branch predictor can be used to predict the address of two basic blocks in a single cycle, improving the hardware ILP of "brainiac" processors. It can also be used to pipeline the address generation process over two cycles in "speed-demon" processors. The prediction accuracy remains the same.

The two-block ahead branch predictor can obviously be extended to a multiple-block ahead branch predictor fetching multiple basic blocks in a single cycle or to further pipeline the address generation process over more than two cycles.

The structures of the two-block ahead BTB and the two-block ahead PT presented in this paper have been directly deduced from existing conventional one-block ahead solutions. We are now investigating specific implementations of those two-block ahead structures. For instance, the double I-fetch implementation of our predictor features a couple of dual-ported memory structures. We are looking at ways to get fast and cost effective structures by taking into account the correlation between basic blocks. We are also looking at ways to adapt cost-effective solutions for one-block ahead BTBs [2, 21] to two-block ahead BTBs.

Acknowledgments

We gratefully acknowledge the help and encouragement of the members of the HPS research group at ACAL, University of Michigan, during our work on this project, in particular, Eric Hao, Sanjay Patel, Daniel Friendly, Lee Hwang Lee, Tse Hao Hsing, Darren Vengroff, and

Professor Yale Patt. They give many inputs and provide critical comments on early versions of the paper.

References

[1] M. Butler and Y. N. Patt, "An Investigation of the Performance of Various Dynamic Scheduling Techniques," *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.

[2] B. Calder and D. Grunwald, "Next Cache Line and Set Prediction," *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[3] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[4] S. Dutta and M. Franklin, "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[5] L. Gwennap, "Comparing RISC Microprocessors," *Proceedings of the Microprocessor Forum*, October 1994.

[6] L. Gwennap, "Digital Leads the Pack with 21164," *Microprocessor Report*, Sept. 94.

[7] L. Gwennap, "PA-8000 Combines Complexity and Speed," *Microprocessor Report*, Vol. 8 Num. 15, 1994.

[8] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report*, Vol. 9 Num. 2, 1995.

[9] W. M. Hwu, Y. N. Patt, "Checkpoint Repair for Out-of-Order Execution Machines," *Proceedings of the 14th Annual International Symposium on Computer Architecture*, June 1987.

[10] W. M. Hwu *et al.*, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, January 1993.

[11] IBM and Motorola, "PowerPC 604 RISC Microprocessor User's Manual," MPR604UMU-01, 1994.

[12] IBM and Motorola, "PowerPC 620 RISC Microprocessor Technical Summary," *Advance Information*, MPR620TSU-1, 1994.

[13] M. Johnson, Superscalar Microprocessor Design, *Prentice-Hall*, 1991.

[14] S. Jourdan, P. Sainrat, and D. Litaize, "Exploring Configurations of Functional Units in an Out-of-Order Superscalar Processor," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[15] S. Jourdan, P. Sainrat, and D. Litaize, "An Investigation of the Performance of Various Instruction-Issue Buffer Topologies," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[16] S. A. Mahlke, *et al.*, "Characterizing the Impact of Predicted Execution on Branch Prediction," *Proceedings of the 27th Annual International Symposium on Microarchitecture*, November 1994.

[17] S. McFarling, "Combing Branch Predictors," *Technical Note TN-36*, DEC-WRL, June 1993.

[18] Mips Technologies Incorporated, "R10000 Microprocessor Product Overview," *Technical Report*, October 1994.

[19] D. N. Pnevmatikatos, M. Franklin, and G. S. Sohi, "Control Flow Prediction for Dynamic ILP Processors," *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993.

[20] S. Simone et al., "Implementation Trade-offs in Using a Restricted Data Flow Architecture in a High Performance RISC Microprocessor," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[21] A. Seznec, "Don't use the page number, but a pointer to it," *to appear in Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[22] J. E. Smith, "A Study of Branch Prediction Strategies," *Proceedings of the 8th Annual International Symposium on Computer Architecture*, May 1981.

[23] J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proceedings of the 12th Annual International Symposium on Computer Architecture*, June 1985.

[24] M. D. Smith, "Tracing with Pixie," *Technical report*, Stanford University, April 1991.

[25] "SPEC 92", *Technical report*, December 1992.

[26] S. Weiss and J. E. Smith, POWER and PowerPC: Principles, Architecture and Implementation, *Morgan Kaufmann Publishers Inc.*, 1994.

[27] T. Yeh, D. T. Marr, and Y. N. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.

[28] T. Yeh, "Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors," *PhD thesis*, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor (MI) , 1993.

[29] C. Young and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction using Branch Correlation," *Proceedings of the 6th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur

INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)

ISSN 0249-6399